PA3: Distributed File System

CSCI 5105 SP25
Phase 1 Due: 4/15 @ 11:59pm
Phase 2 Due: 4/22 @ 11:59pm

Overview

In this assignment, you will implement a simple distributed file system capable of hosting multiple clients that can share files together seamlessly. Files will be replicated to several file servers within the system following the quorum replication protocol (Gifford algorithm for replication). Clients will be able to communicate with any server within the system to read or write to any file stored by the system.

Description

Our file system will be made from two main components: a **client** and a **replica server**. You will be responsible for implementing each component of the system.

Replica servers will be started as part of a network of similar replica servers. Each replica will have its own directory to store files locally. All replicas will be able to communicate between one another, with one server designated as the coordinator. The coordinator will manage consistency and control the read / write quorums of your system - a replica which receives a read / write request to a file must forward the request to the coordinator. It will then verify a file's version status by checking the file's current version on a randomly selected number of nodes following the size requirements of the read / write quorums before processing the client's original request.

The **client** should be able to contact any replica server within the system to make read and write requests. It should be able to access, read or write, to any files within the system, even if the replica the client contacts does not physically store the file requested. The client must also be able to list all files (with versions) from within the system, also including files not stored locally by the replica which it contacted.

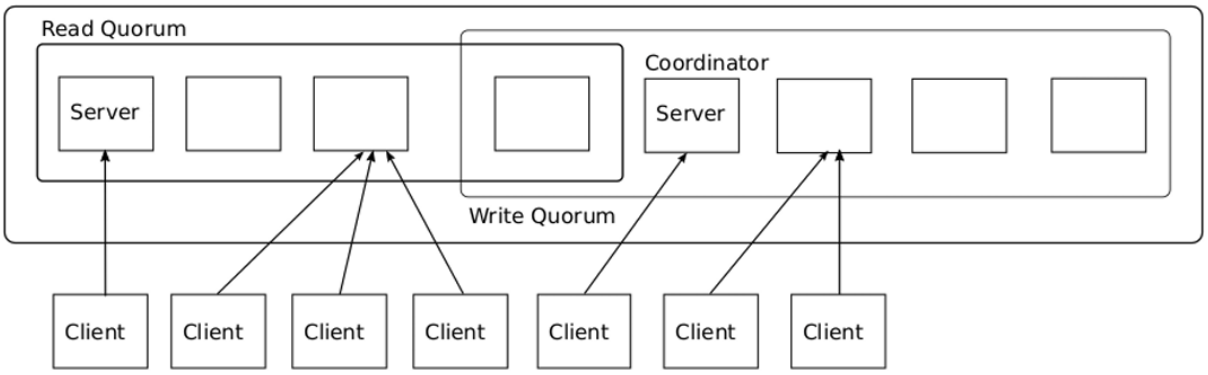
Quorum Protocol

A quorum is "the minimum number of members of a group necessary to constitute the group at a meeting." For our filesystem, A read and write quorum (group of replica servers) will be made up of arbitrarily selected nodes and managed by the coordinator to verify the most recent version of a file when requested. Nodes within each quorum will provide 'file versions' upon request such that the only files read or written to are the most up-to-date versions of each file. The number of replica servers within a quorum must follow the below two rules:

1. $N_R + N_W > N$
2. $N_W > N / 2$

Where N_R is the number of replicas within the read quorum, N_W is the number of replicas within the write quorum, and N is the total number of replicas within the system. Guaranteeing these quorum sizes will produce read and write quorums where at least one replica server lies within both quorums. Overlap among the two quorums guarantees that any read or write always occurs on the most recent version of a file.

Both read and write requests, whenever received by a replica, must be forwarded to the coordinator. The coordinator should then arbitrarily select replicas to fill the corresponding quorum, and query each selected replica to determine the highest version number of a file. The highest version number should be returned to the original replica to be used in processing the original request. Your coordinator must RANDOMLY select replicas to fill each quorum upon request, with quorum sizes determined by the two rules defined above and initialized in your 'compute_nodes.txt' file (details on 'compute_nodes.txt' in a future section).



Managing Files

Files will be managed by their 'version number' - a number associated with a file that is incremented each time the file is written to. Replicas should track the version numbers of files contained within their local directories. New files within the system should be initialized with version number 0, with following writes to existing files increasing the file's version count by 1. When a replica downloads a file from another replica, its local copy should inherit the version number of the file of the replica it originated from. You may assume file versions are immutable, such that performing a write overwrites old copies of the file. The file system should implement both read and write operations for use on any file format, including .txt, .png, etc following quorum protocols.

System Architecture: Replica Server

Replicas will provide clients with read / write access to all files within the filesystem, including files stored 'locally' in the replica's local directory and files stored across the network. You will be responsible for designing and implementing the RPC interface and functionality your replicas will use.

File Storage

Replicas should have a local directory in which they can store files from the system. A replicas should be able to set the path to its local directory on startup via command line argument. Each replica is assumed to have exclusive access to its own local directory and should be initialized with a local directory no other replica is initialized to. It is assumed that all files within this directory are contained at the same level, i.e. no subdirectories exist. You may also assume this directory is empty when the replica is started.

Read Requests

A client can make a read request by sending the name of a file to any replica within the system. The replica, following quorum protocol, must provide the client access to the requested file by returning a path to the file stored within the replica's local directory. If the file does not exist within the system, the client should receive an error response. The workflow for a read request should be as follows:

1. A client makes a request to read file F from any replica R in the system.
2. Replica R forwards the request to the coordinator C.
3. C queues and eventually processes the request. C finds the highest version of F from querying replicas within the randomly chosen read quorum.
4. C returns the highest version number of F to R.
    a. If R has F with the same version as the one returned by C, R provides the client access to its local copy of F.
    b. If R does not have the same version of F as returned by C, R must copy the file with the version returned by C to its local directory from the replica which contains it. R will then provide access to the client of its local copy of F.

Write Requests

A client can make a write request by sending the filepath of the file to write to any replica within the system. The receiving replica should copy this file to its local directory, setting its version number following quorum protocol. The replica's new version of the file should be updated in the system by copying the new file to all other replicas within write quorum. The client should receive a message from the replica indicating whether or not its request was processed successfully. The workflow for a write request is as follows:

1. A client makes a request to write file F from any replica R in the system.
2. Replica R forwards the request to coordinator C.
3. C queues and eventually processes the request. C finds the highest version of F from querying nodes within the randomly chosen write quorum.
4. C returns the highest version of F to R. R writes a new version of F with version 1 + C's returned version to its local directory.
5. The coordinator should tell all other replicas within the write quorum to copy R's newest version of F to their local directories after R has completed its write. If a replica already has a lower version of the file, it is simply overwritten with the newest version.

Inter-node Communication

Servers within the network will know the connection information (ip,port) of all other servers using a compute_nodes.txt file. This file will be similar to previous projects with a new first line and added argument to each replica address. The first line should indicate the sizes of your quorums N_R, N_W such that the coordinator can read this line to determine the number of nodes to make up each of the read and write quorums. Following entries should contain connection information as `ip,port,flag` where flag is 1 if the node at `ip,port` is the coordinator, and 0 if it is not (there should only be one flag set to 1 in the entire file). Replicas within the system can read this file to communicate with other nodes and the coordinator. You may assume that each address in this file corresponds to an active replica server.

File Transfer

When transferring files between servers, files should be transferred in chunks. Files may be large in size; thus, transferring a large file in one thrift response is not a good approach. A function should be provided in your replica that allows requesting clients to retrieve a chunk of a file specified by the size and offset within a file - file chunks sent through thrift for this project should be no larger than 2048 bytes. You will need to read / write files in binary and send binary data through thrift with thrift's 'binary' type since files may be of any type. You may find another thrift function to return the size of a file to be helpful for clients attempting to download a file.

Replica Server: Coordinator

One replica server within the file system will act as a coordinator, determined at startup. The coordinator must be capable of managing read / write requests from other replica servers within the network following quorum protocol. When the coordinator processes a read or write request, it must check with the respective quorum of replicas before attempting to process the request. The coordinator must randomly select a set of replicas upon each read / write request to fill each quorum. The sizes of N_R and N_W should be initialized in the first line of your 'compute_nodes.txt' file as N_R, N_W and should follow the quorum size rules of $N_R + N_W > N$ and $N_W > N / 2$.

The coordinator will also be responsible for managing consistency throughout the system. Read and Write requests forwarded from replicas should be placed within a queue. This queue should guarantee sequential consistency between reads and writes across the system by only allowing 1 request to execute at a time.

System Architecture: Client

The client should be able to query any replica server within the filesystem to make read and write requests to any files stored by the system. Read requests should pass the name of a file to be read by the client and return the path to the file stored within the local directory of the replica the client initially contacted. Write requests should pass the file path of a file to write to the system, then return a message to the client indicating the status of the write. The client should also provide functionality to list the files and version numbers of ALL files stored across the system. Your client should be implemented as a simple command line program. One evocation of your client ('./client' or 'python client') should process 1 read / write / list request.

Assumptions

- Random server / network failure is not a concern
- Replicas will know the ip and port of other replicas and the coordinator by using compute_nodes.txt. Each address can be assumed to link to an online node.
- Files may be of any type or size.
- The system should be capable of handling up to 64 total files
- All files stored by a replica will be at the same level within the replica's local directory; the system will not need to manage sub-directories.

You can make other assumptions within reason so long as you define them in your submission's design document

Hints

Check the canvas page for an updated list of hints

Submission: Design Document

Your submission should include a design document that contains a description of the functions your replica servers provide. Additionally, your document should include evaluations of the performance of requests to your network with changing quorum sizes. You must include timings for evaluating your network with at least 7 nodes over a range of N_R and N_W values. Record the timing results for the system when >=3 clients concurrently request to read (read heavy workload) or write (write heavy workload). Find which combinations of N_R and N_W work best for each workload and support your findings with timing data. Visualize the results of your timings through a graph, table, or similar.

This document will be worth more of the total project grade (35%) than in previous projects and should contain more detail in regards to testing. Please spend a proportionate amount of time testing and documenting your system using the parameters described above.

Submission: Deliverables

You will submit a zip file to gradescope containing the following components:

- Design document
- README.txt - this file should describe how to compile and execute each component of your system. This should be separate from the design document.
- Compute_nodes.txt
- Source code for each of your components
- Thrift files for each of your components
    - Thrift generated code is not necessary, but may be submitted as well

Your submission should only include the above files! Do not include .git, _MACOSX, object files, and any other files which are not listed above. Your project should be able to be compiled

from the files above. Please double check your submission has all components before finalizing your submission. Some points will be lost for not following submission guidelines!

Submission: Other Logistics

Groups - you may work in groups of max two for this project. Make sure whoever submits your work includes the other partner's email when submitting on gradescope.

Grading - the grading breakdown will be as follows:

Phase 1 (10%) - File transfer functionality. Implement file transfer capabilities for your replica server, and have a replica copy a file from another replica. Submit a brief document describing the progress you have made on the project thus far and what challenges lie ahead. ALSO, submit the code for your replica with file transfer functionality.

Phase 2 (90%) - Final submission, see the below grade breakdown:

- Design Document (35%) - see above for details
- System functionality (45%) - You will lose points if your system has any exceptions, crashes, or freezes.
- Source code quality (10%) - Manual check of source code quality and style.

Use of AI Tools

Artificial intelligence (AI) generative tools may be used in a **limited manner for programming assignments with appropriate attribution and citation**. For programming assignments, you may use AI tools to help with some basic helper function code (similar to library functions). You must clearly identify parts of the code that you used AI tools for, providing a justification for doing so. You must understand such code and be prepared to explain its functionality.

You must not use AI tools to design your solution, or to generate a significant part of your assignment code. You must design and implement the code yourself. Any report for the assignment must be written yourself without using AI tools.

Any use of AI-generated code without attribution or justification, or if it is considered to form a significant part of your design and implementation, would be considered cheating.

Note that the goal of these assignments is to provide you with a deeper and hands-on understanding of the concepts you learn in the class, and not merely to produce "something that works". Hence a partially-working solution is preferable to a fully-functioning solution built with substantial use of AI.

If you are in doubt as to whether you are using AI language models appropriately in this course, I encourage you to discuss your situation with me. Examples of citing AI language models are available at: libguides.umn.edu/chatgpt. You are responsible for fact checking statements and correctness of code composed by AI language models.